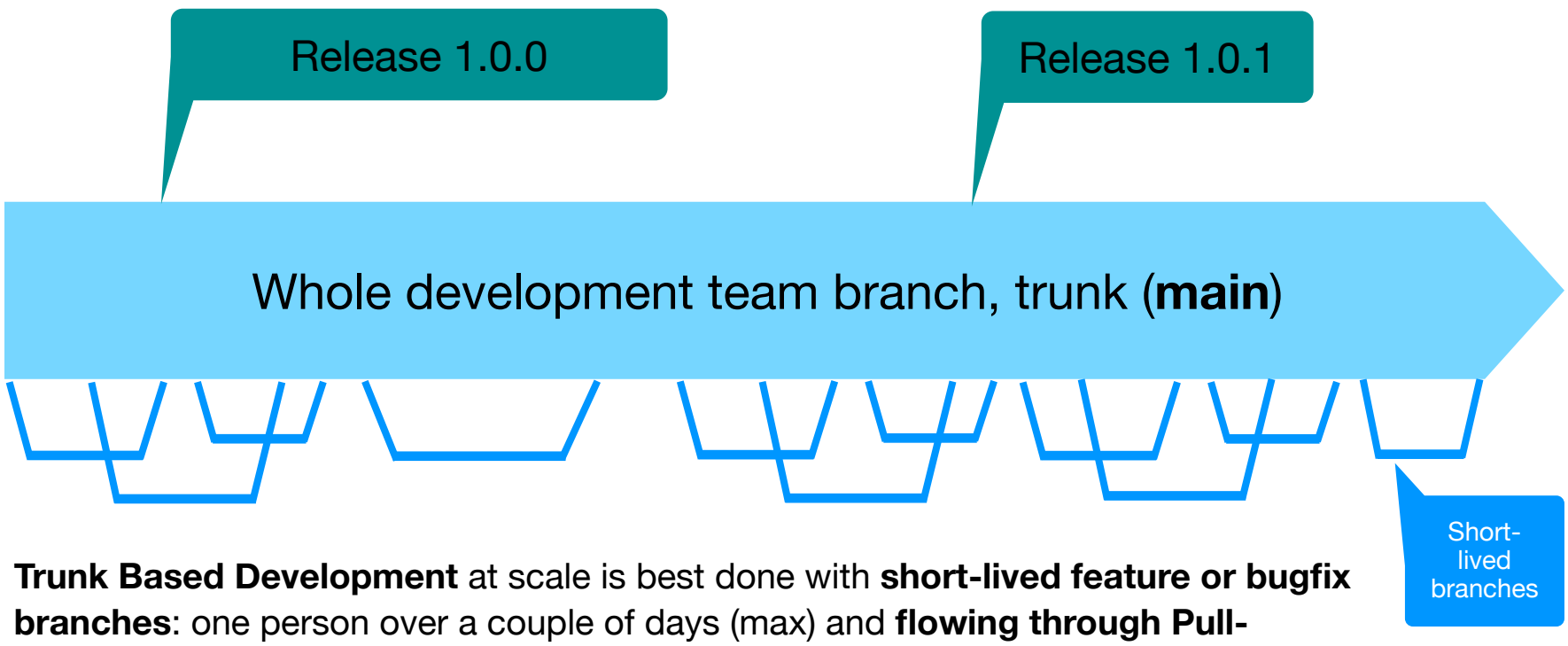
Applied CICD Patterns

The purpose of this document is to provide a strategy for **source code integration** based on Trunk Based Development and on top of that to define a **continuous delivery pattern**, from **development environment** to **production releases**. Developers can be considered as anyone contributing code, including Devs, SRE/DevOps, QAs, Architects, etc.

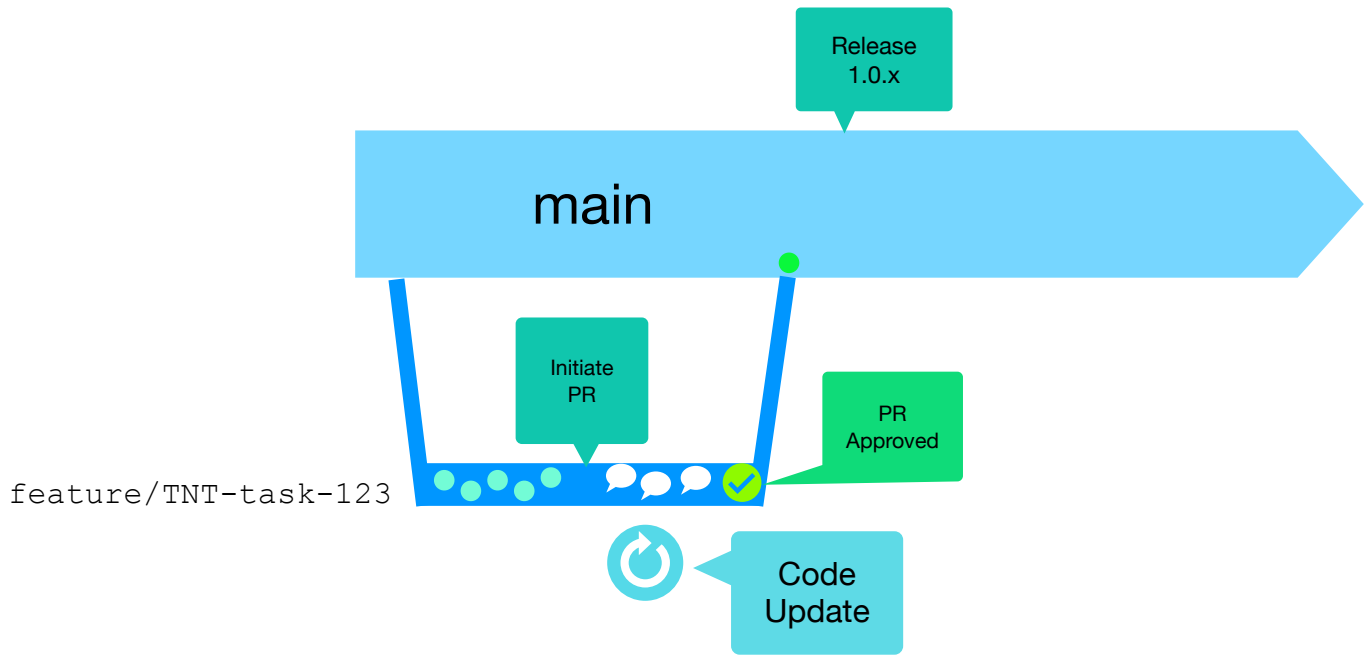
Continuous Integration

Trunk Based Development is a source-control branching model, where developers collaborate with code into a single branch called **trunk** (**master** or **main**, based on Git nomenclature). Developers create very short lived feature-branches that gets merged regularly into the trunk through **pull requests** that includes a peer review (**PR**), then by using a **development pipeline automation** to ensure at least, that the merge will not break the build, the code is secure and of good quality.



Trunk Based Development at scale is best done with **short-lived feature or bugfix branches**: one person over a couple of days (max) and **flowing through Pull-Request** style code review and **build automation** before integrating (merging) into the trunk

Example of a **short-lived** feature branch and PR merge.



In the picture above, the branch `feature/TNT-task-123` is opened from **main**, then commits are added into it in a regular basis, then a PR is initiated, usually some comments and reviews will happen that will imply pushing new code until the approval is done, usually for at least one peer of the team, then the merge is done into **main** and the feature branch is deleted from the code base.

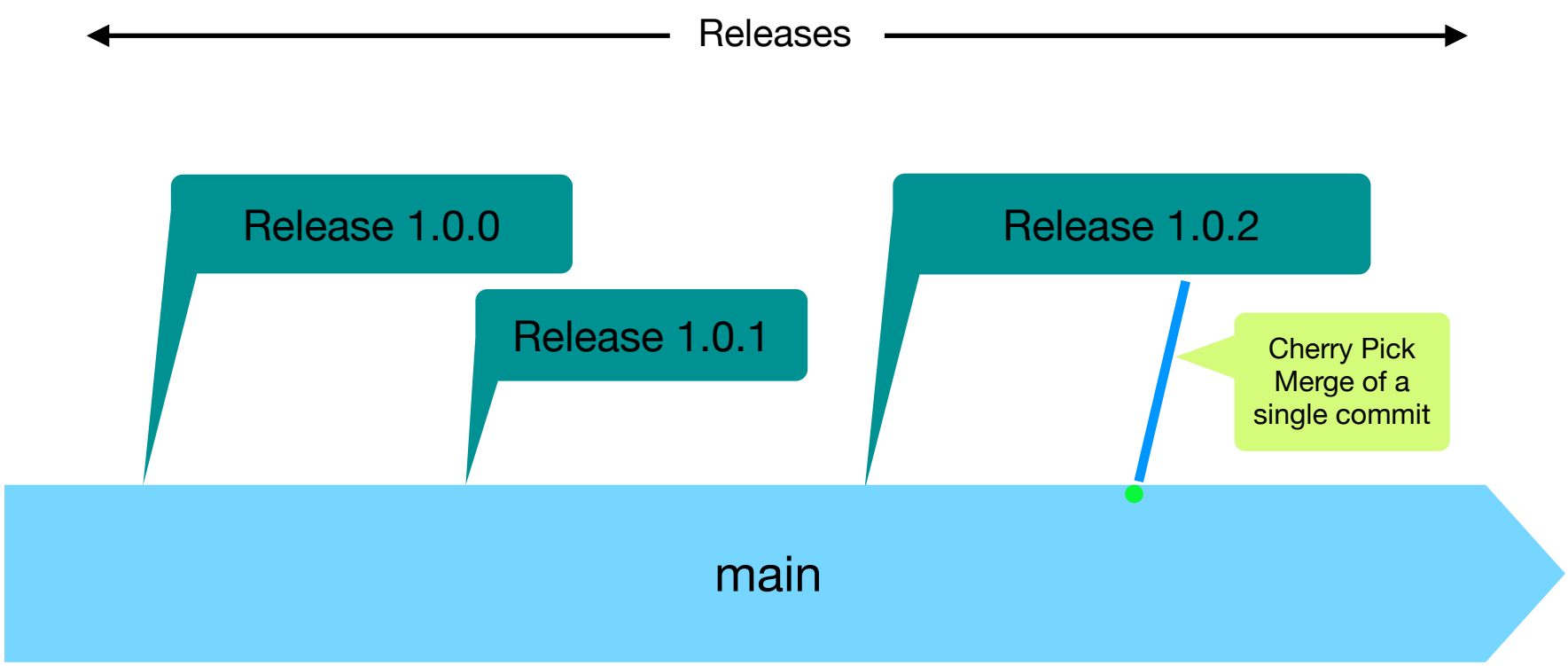
Proposed branch **naming** and source integration **flow**:

- **main:** For the Trunk branch, commits are not done directly to the main, its **always** via a peer review process.
- **feature|bugfix/<project_key>-<task_number>:** For any feature or bugfix that is new or to update as part of the related task, this imply updating pure *app code*, *tests*, *documentation*, *infrastructure as code*, *pipeline definition*, etc.
  - **<project_key>:** Correspond to a *key* or *abbreviation* for the project usually in capital case.
  - **<task_number>:** Correspond with the minimal task unit on an Agile board that is considered a *1:1 relation* with the *feature* or *bugfix* that is going to be incorporated as part of the development work. These **tickets** could have the following order of precedence:
    - **Epic > Feature > User Story > (Task, Bug)**

    - **Feature:** This ticket might have several User Stories
    - **User Story:** It could have several *tasks* and *bugfix* related, and in this case the branches will be created based on the task or bugfix number, **i.e.** for the **TNT** project, **feature/TNT-task-123** **bugfix/TNT-bug-456**

- **release-<version>:** Is the release branch that is created from the **main** trunk, following the [semantic versioning] as candidate for **production release**, this could be the entire set of tasks of an sprint, usually 2 weeks of work, or a set of those tasks agreed during the planning that could be used for a release, this way the pattern allow the following **release cycle**:
    - **Full Sprint Release:** Making a release candidate the entire set of `feature|bugfix` added on the sprint.
    - **Partial Sprint Release:** A set of `feature or bugfix` that can be branched out as release branch for release candidate. This can also be for single commit release candidates, such practice allows faster **production** deployments during a sprint.

In such release branch, whether if its the full sprint, a set of features or a single feature, testing processes (QA, UAT, etc.) For **automated** and **non-automated** needs to happen on this branch, to certify a release candidate prior to the **production** deployment.



As shown in the picture, in the event that a *bugfix* has been done for an specific release branch, then such commit needs to be added

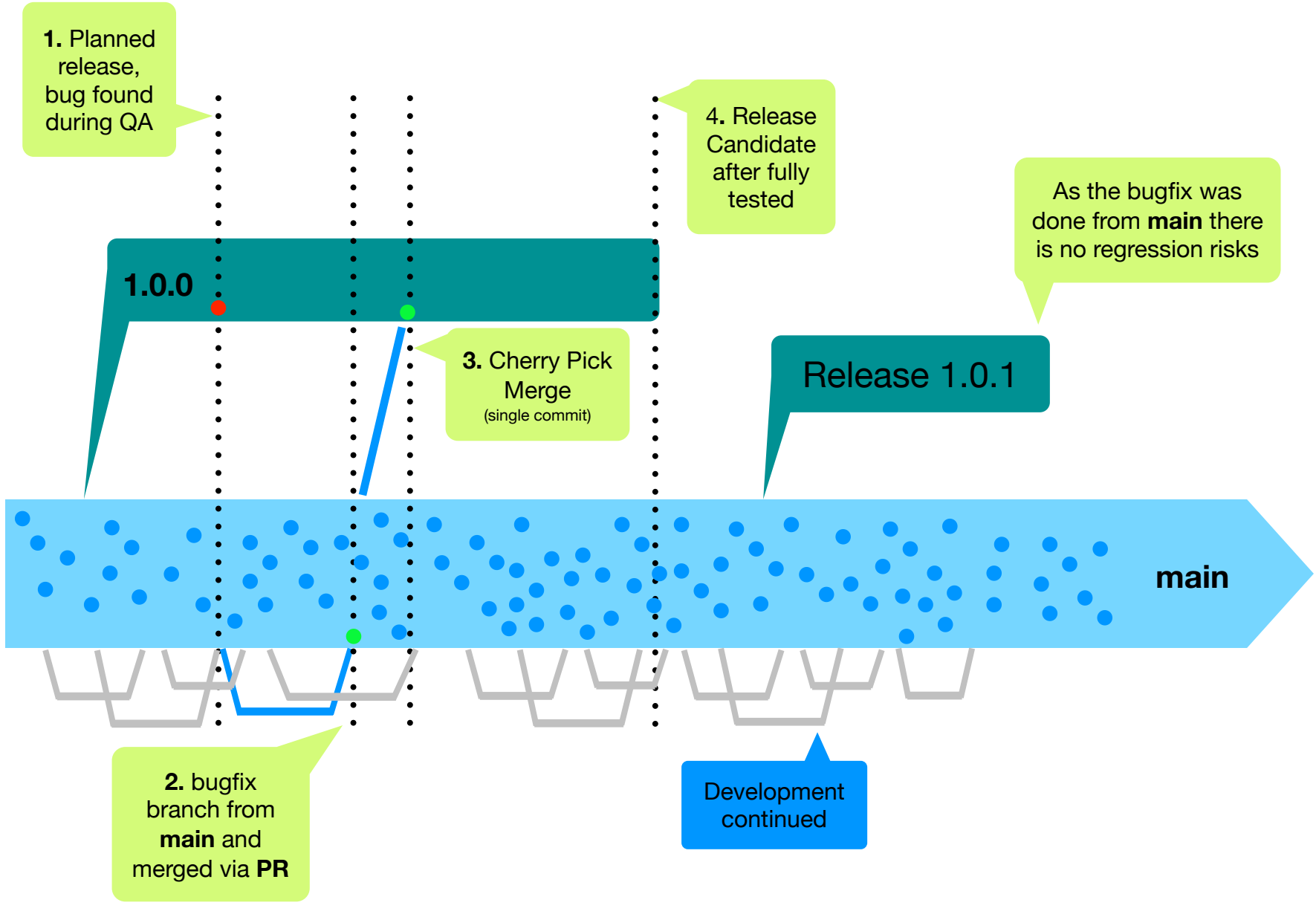
into the release branch via [Git Cherry Pick](), bellow is pretty much what a `bugfix` flow will be:

1. **branch out** from current **main** that will be at the same point in time or more since the release branch was initially created, usually the state of **main** when the bugfix branch is created will have more context into it, as the *development team* continued since the release branch have been initially tested.

2. Once the development of the bugfix has **finished**, a peer review will be opened to get it merged into **main**.

3. Once merged, a **cherry pick** process will add such commit into the specific release branch.
   Note: Either release branch creation and/or cherry pick process, can be automated, such automation can be added later after subsequent sprints that shows that the CICD pattern is giving the expected results, similar to PR merges into **main** after full approval.

Bellow picture shows the **dynamics** of a bugfix on a planned release.



Quote from Google “*So, a release branch is typically a snapshot from trunk with an optional number of cherry picks that are developed on trunk and then pulled into the branch*”.

One of the principles of Trunk based development is **No commit pushed to the trunk should jeopardise the ability to go live**.

Iterations needs to happen before gaining reliability and some degree of security, which is based on **code coverage**, plus all the different set of **tests** related with the application itself as the infrastructure, to release them to production on short cycles.

Continuous Delivery